HPC: HW3 Report

2017-19841 최창민

1. Computing pi with MPI

```
double monte_carlo(double *xs, double *ys, int num_points, int mpi_rank, int
mpi_world_size, int threads_per_process) {
  int count = 0;

  // TODO: Parallelize the code using mpi_world_size processes (1 process
per
  // node.
  // In total, (mpi_world_size * threads_per_process) threads will
collaborate
  // to compute pi.
  int local_count = 0;
  omp_set_num_threads(threads_per_process);
  int step = num_points / mpi_world_size;
  int last_step = 0;
  MPI_Bcast(xs, num_points, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast(ys, num_points, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  #pragma parallel for
  for (int i = mpi_rank * step; i < MIN((mpi_rank + 1) * step, num_points);
i++) {
    double x = xs[i];
    double y = ys[i];
    if (x*x + y*y <= 1)
      local_count++;
  }

  MPI_Reduce(&local_count, &count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

  // Rank 0 should return the estimated PI value
  // Other processes can return any value (don't care)
  return (double) 4 * count / num_points;
}
```

병렬화 구현 방식

xs와 ys를 노드 전부에 배분한 다음 각 노드에서 1/4씩 나누어서 openmp를 사용하여 실행하였다. 이때 xs와 ys를 좀 더 효율적으로 필요한 부분만 배분할 수도 있지만, 병렬화의 속도는 측정하지 않기 때문에

간단하게 Bcast를 사용하여 값을 모든 노드에 배분하였다. 이후 계산된 값들을 Reduce를 사용하여 모은 다음, pi 계산식에 따라서 계산하여 return하였다.

2. 행렬곱 챌린지

설명에 필수적인 부분만 남긴 코드

```
  void transpose_single(float *src, float *dst, const int I, const int J, const
int block_size) {
    // I x J -> J x I
    for (int i = 0; i < block_size; ++i) {
      for (int j = 0; j < block_size; ++j) {
        dst[i + I * j] = src[j + J * i];
      }
    }
  }

  void transpose(float *src, float *dst, const int I, const int J, const int
block_size) {
    // I x J -> J x I
    #pragma omp for
    for (int i = 0; i < I; i += block_size) {
      for (int j = 0; j < J; j += block_size) {
        transpose_single(&src[j + J * i], &dst[i + I * j], I, J, block_size);
      }
    }
  }

  __m512 __vectordot(float *A, float *B, const int K, const int mpi_world_size)
{
    for (int k = 0; k < K; k += 16) {
      a = _mm512_loadu_ps(&A[k]);
      b = _mm512_loadu_ps(&B[k]);
      curr = _mm512_fmadd_ps(a, b, curr);;
    }
    return curr;
  }

  void matmul(float *A, float *B, float *C, int M, int N, int K,
      int threads_per_process, int mpi_rank, int mpi_world_size) {
    // A: M x K
```

```
    // B: K x N
    // C: M x N
    MPI_Bcast(B, K * N, MPI_FLOAT, 0, MPI_COMM_WORLD);
    MPI_Scatter(A, chunkM * K, MPI_FLOAT, Ac, chunkM * K, MPI_FLOAT, 0,
MPI_COMM_WORLD);
    #pragma omp parallel private(curr)
    {
      transpose(B, Bp, K, N, tp_sz);
      #pragma omp for
      for (int mm = 0; mm < chunkM; mm += sz) {
        for (int nn = 0; nn < N; nn += sz) {
          for (int m = mm; m < MIN(mm + sz, chunkM); ++m) {
            for (int n = nn; n < MIN(nn + sz, N); ++n) {
              curr = __vectordot(&Ac[K * m], &Bp[K * n], K, mpi_world_size);
              Cc[n + N * m] = _mm512_reduce_add_ps(curr);
            }
          }
        }
      }
    }
    MPI_Gather(Cc, chunkM * N, MPI_FLOAT, C, chunkM * N, MPI_FLOAT, 0,
MPI_COMM_WORLD);
  }
```

병렬화 방식에 대한 설명

병렬화를 위해서 3개의 외부 함수를 구했습니다.

첫번째는 vector dot함수로 AVX512의 FMA함수를 써서 벡터의 내적을 하는 함수를 구현했습니다. float로 인풋을 받아서 m512로 아웃풋을 내보내게 했습니다.

두번째와 세번째는 transpose함수입니다. traspose는 A x B에서 B 행렬을 transpose하여 기존에 A는 가로로 연산이 진행되고 B는 세로로 연산이 진행되던 것을 B연산 또한 가로로 진행되도록 하여서 for루프를 돌 때 좀 더 cache hit을 높이려고 하였습니다. openmp를 사용하여 병렬화했고 단순히 행렬 전부를 for loop 돌며 transpose하게 되면 속도가 느려서 tiling을 하여서 속도를 높였습니다. tiling을 할 때 transpose 한 block을 모아서 다시 transpose하였는데 이는 그림으로 표현하면 다음과 같습니다.

(2, 1)사이즈 블록 1 2 | 3 4 | 5 6 ——————— 7 8 | 9 10 | 11 12	=>	블록별로 transpose 1 | 3 | 5 2 | 4 | 6 ———— 7 | 9 | 11 8 | 10 | 12	=>	블록마다 transpose 1 | 7 2 | 8 3 | 9 4 | 10 5 | 11

메인 함수에서는 B행렬은 모든 노드에 Bcast로 전부 전달하고 A행렬은 Scatter로 필요한 만큼 쪼개어서 전달하였습니다. 이는 B행렬은 세로 방향로 잘린 것이 필요하기 때문에 쪼개서 전달할 수 없었고 A행렬은 가로로 잘라진 것이 필요하기 때문에 Scatter 함수를 통해 잘라서 전달할 수 있었습니다. 각 노드들은 전달받은 B를 transpose하여 사용했고 openmp를 사용하여 병렬화를 하면서 여기에도 tiling을 사용하여 속도를 높였습니다. 이렇게 연산한 결과값들을 MPI_Gather를 통해서 값들을 모았습니다.

최적화에 적용한 방법 및 고려사항

A행렬을 세로로, B행렬을 가로로 자르는 방식 등 여러가지 방식으로 자르고 연산하는 것을 적용해보았는데, 그 과정에서 알게된 것이 transpose연산이 느리다는 것을 알게되었습니다. 그래서 tiling을 transpose에 적용할 수 있으면 좋겠다는 생각을 했고 Block단위로 transpose한 것을 다시 transpose하더라도 전체 값을 transpose한 것과 같다는 것을 알게 되고 이를 구현하였습니다.

AVX512연산에 대해서도 연산 목록을 살펴보면서 고민했었는데, vector dot을 구현하는 것은 크게 어렵지 않았지만 이를 구현할 때 A,B 행렬을 어떻게 불러올 지, C행렬로 저장할 지 고민을 하였습니다. 처음에는 한번에 다 불러와서 배열로 저장하고 C행렬을 m512로 저장했다가 한번에 변환하는 것을 생각하였지만 크게 속도차이가 안 나거나 오히려 더 느렸던 것으로 기억하고 있습니다. 또한 이렇게 저장할 때 m512를 float로 바꿀 때 처음에는 제가 직접 해야하는 것으로 생각했었지만 찾아보니 해당하는 연산이 있어서 이를 적용했습니다.

통신 패턴에 대한 설명

저는 Scatter, Bcast, Gather를 사용했습니다. Scatter는 rank마다 지정된 크기의 block을 겹치지 않고 특정 rank에서 균등하게 배분하는 방식이고, Bcast는 해당하는 크기를 특정 rank에서 노드 전부에 전달하는 방식입니다. Gather는 Scatter과 비슷하지만 반대의 방향으로 rank마다 지정된 크기의 block을 특정 rank로 전달주는 방식입니다. 이렇게 전달받으면 rank순으로 연결되어집니다.

M, N, K가 2의 지수승이 아닐 때 strip-mining의 복잡성에 대한 논의

2의 지수승이 아닐 경우 다양한 엣지 케이스를 핸들링해야 합니다. 이는 보통 memory align이나 cache의 단위 등이 대부분 2의 지수승이기 때문에 strip size도 2의 지수승으로 설정하게 되는데 이 경우 M, N, K가 2의 지수승 임을 가정하지 않으면 메모리 접근 등에서 나누기 나머지 연산등으로 엣지케이스들을 핸들링해줘야합니다.

또한 strip size로 딱 나누어떨어지지 않을 수 있고 이것 때문에 일부분이 낭비되는 tile이 발생하게 됩니다. 비슷하게 SIMD에서도 AVX512 연산에서 512비트 중 일부분이 낭비되는 현상이 발생할 수 있습니다. 이러한 현상은 오히려 효율성을 감소시킬 수 있기 때문에 이런 경우에 효율적으로 작성하기 복잡합니다.